The Pragmatic Programmers

Language Implementation Patterns

Create Your Own Domain-Specific and General Programming Languages

Terence Parr

Edited by Susannah Davidson Pfalzer

What Readers Are Saying About
Language Implementation Patterns

Throw away your compiler theory book! Terence Parr shows how to write practical parsers, translators, interpreters, and other language applications using modern tools and design patterns. Whether you're designing your own DSL or mining existing code for bugs or gems, you'll find example code and suggested patterns in this clearly written book about all aspects of parsing technology.

➤ **Guido van Rossum**
Creator of the Python language

My Dragon book is getting jealous!

➤ **Dan Bornstein**
Designer, Dalvik Virtual Machine for the Android platform

Invaluable, practical wisdom for any language designer.

➤ **Tom Nurkkala, PhD**
Associate Professor, Computer Science and Engineering, Taylor University

Terence makes language design concepts clear and approachable. If you ever wanted to build your own language but didn't know where to start or thought it was too hard, start with this book.

➤ **Adam Keys**
http://therealadam.com

This is a book of broad and lasting scope, written in the engaging and accessible style of the mentors we remember best. *Language Implementation Patterns* does more than explain how to create languages; it explains how to *think* about creating languages. It's an invaluable resource for implementing robust, maintainable domain-specific languages.

➤ **Kyle Ferrio, PhD**
Director of Scientific Software Development, Breault Research Organization

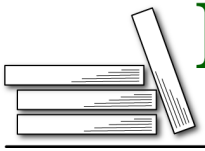
Language Implementation Patterns

Create Your Own Domain-Specific and General Programming Languages

Terence Parr

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-934356-45-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P5.0—September 2014

Contents

Part II — Analyzing Languages

Part III — Building Interpreters

Part IV — Translating and Generating Languages

Acknowledgments

I'd like to start out by recognizing my development editor, the talented Susannah Pfalzer. She and I brainstormed and experimented for eight months until we found the right formula for this book. She was invaluable throughout the construction of this book.

Next, I'd like to thank the cadre of book reviewers (in no particular order): Kyle Ferrio, Dragos Manolescu, Gerald Rosenberg, Johannes Luber, Karl Pfalzer, Stuart Halloway, Tom Nurkkala, Adam Keys, Martijn Reuvers, William Gallagher, Graham Wideman, and Dan Bornstein. Although not an official reviewer, Wayne Stewart provided a huge amount of feedback on the errata website. Martijn Reuvers also created the ANT build files for the code directories.

Gerald Rosenberg and Graham Wideman deserve special attention for their ridiculously thorough reviews of the manuscript as well as provocative conversations by phone.

Preface

The more language applications you build, the more patterns you'll see. The truth is that the architecture of most language applications is freakishly similar. A broken record plays in my head every time I start a new language application: "First build a syntax recognizer that creates a data structure in memory. Then sniff the data structure, collecting information or altering the structure. Finally, build a report or code generator that feeds off the data structure." You even start seeing patterns within the tasks themselves. Tasks share lots of common algorithms and data structures.

Once you get these language implementation design patterns and the general architecture into your head, you can build pretty much whatever you want. If you need to learn how to build languages pronto, this book is for you. It's a pragmatic book that identifies and distills the common design patterns to their essence. You'll learn why you need the patterns, how to implement them, and how they fit together. You'll be a competent language developer in no time!

Building a new language doesn't require a great deal of theoretical computer science. You might be skeptical because every book you've picked up on language development has focused on compilers. Yes, building a compiler for a general-purpose programming language requires a strong computer science background. But, most of us don't build compilers. So, this book focuses on the things that we build all the time: configuration file readers, data readers, model-driven code generators, source-to-source translators, source analyzers, and interpreters. We'll also code in Java rather than a primarily academic language like Scheme so that you can directly apply what you learn in this book to real-world projects.

What to Expect from This Book

This book gives you just the tools you'll need to develop day-to-day language applications. You'll be able to handle all but the really advanced or esoteric

situations. For example, we won't have space to cover topics such as machine code generation, register allocation, automatic garbage collection, thread models, and extremely efficient interpreters. You'll get good all-around expertise implementing modest languages, and you'll get respectable expertise in processing or translating complex languages.

This book explains how existing language applications work so you can build your own. To do so, we're going to break them down into a series of well-understood and commonly used patterns. But, keep in mind that this book is a learning tool, not a library of language implementations. You'll see many sample implementations throughout the book, though. Samples make the discussions more concrete and provide excellent foundations from which to build new applications.

It's also important to point out that we're going to focus on building applications for languages that already exist (or languages you design that are very close to existing languages). Language design, on the other hand, focuses on coming up with a syntax (a set of valid sentences) and describing the complete semantics (what every possible input means). Although we won't specifically study how to design languages, you'll actually absorb a lot as we go through the book. A good way to learn about language design is to look at lots of different languages. It'll help if you research the history of programming languages to see how languages change over time.

When we talk about language applications, we're not just talking about *implementing* languages with a compiler or interpreter. We're talking about any program that processes, analyzes, or translates an input file. Implementing a language means building an application that executes or performs tasks according to sentences in that language. That's just one of the things we can do for a given language definition. For example, from the definition of C, we can build a C compiler, a translator from C to Java, or a tool that instruments C code to isolate memory leaks. Similarly, think about all the tools built into the Eclipse development environment for Java. Beyond the compiler, Eclipse can refactor, reformat, search, syntax highlight, and so on.

You can use the patterns in this book to build language applications for any computer language, which of course includes domain-specific languages (DSLs). A domain-specific language is just that: a computer language designed to make users particularly productive in a specific domain. Examples include Mathematica, shell scripts, wikis, UML, XSLT, makefiles, PostScript, formal grammars, and even data file formats like comma-separated values and XML. The opposite of a DSL is a general-purpose programming language like C, Java, or Python. In common usage, DSLs also typically have the connotation

of being smaller because of their focus. This isn't always the case, though. SQL, for example, is a lot bigger than most general-purpose programming languages.

How This Book Is Organized

This book is divided into four parts:

- *Getting Started with Parsing*: We'll start out by looking at the overall architecture of language applications and then jump into the key language recognition (parsing) patterns.
- *Analyzing Languages*: To analyze DSLs and programming languages, we'll use parsers to build trees that represent language constructs in memory. By walking those trees, we can track and identify the various symbols (such as variables and functions) in the input. We can also compute expression result-type information (such as `int` and `float`). The patterns in this part explain how to check whether an input stream makes sense.
- *Building Interpreters*: This part has four different interpreter patterns. The interpreters vary in terms of implementation difficulty and run-time efficiency.
- *Translating and Generating Languages*: In the final part, we will learn how to translate one language to another and how to generate text using the StringTemplate template engine. In the final chapter, we'll lay out the architecture of some interesting language applications to get you started building languages on your own.

The chapters within the different parts proceed in the order you'd follow to implement a language. Section 1.2, *A Tour of the Patterns*,on page 4 describes how all the patterns fit together.

What You'll Find in the Patterns

There are 31 patterns in this book. Each one describes a common data structure, algorithm, or strategy you're likely to find in language applications. Each pattern has four parts:

- *Purpose*: This section briefly describes what the pattern is for. For example, the purpose of Pattern 21, *Automatic Type Promotion*,on page 190 says "…how to automatically and safely promote arithmetic operand types." It's a good idea to scan the Purpose section before jumping into a pattern to discover exactly what it's trying to solve.

- *Discussion*: This section describes the problem in more detail, explains when to use the pattern, and describes how the pattern works.
- *Implementation*: Each pattern has a sample implementation in Java (possibly using language tools such as ANTLR). The sample implementations are not intended to be libraries that you can immediately apply to your problem. They demonstrate, in code, what we talk about in the Discussion sections.
- *Related Patterns*. This section lists alternative patterns that solve the same problem or patterns we depend on to implement this pattern.

The chapter introductory materials and the patterns themselves often provide comparisons between patterns to keep everything in proper perspective.

Who Should Read This Book

If you're a practicing software developer or computer science student and you want to learn how to implement computer languages, this book is for you. By computer language, I mean everything from data formats, network protocols, configuration files, specialized math languages, and hardware description languages to general-purpose programming languages.

You don't need a background in formal language theory, but the code and discussions in this book assume a solid programming background.

To get the most out of this book, you should be fairly comfortable with recursion. Many algorithms and processes are inherently recursive. We'll use recursion to do everything from recognizing input, walking trees, and building interpreters to generating output.

How to Read This Book

If you're new to language implementation, start with Chapter 1, *Language Applications Cracked Open*,on page 3 because it provides an architectural overview of how we build languages. You can then move on to Chapter 2, *Basic Parsing Patterns*,on page 19 and Chapter 3, *Enhanced Parsing Patterns*,on page 47 to get some background on grammars (formal language descriptions) and language recognition.

If you've taken a fair number of computer science courses, you can skip ahead to either Chapter 4, *Building Intermediate Form Trees*,on page 71 or Chapter 5, *Walking and Rewriting Trees*,on page 99. Even if you've built a lot of trees

and tree walkers in your career, it's still worth looking at Pattern 14, *Tree Grammar*,on page 117 and Pattern 15, *Tree Pattern Matcher*,on page 120.

If you've done some basic language application work before, you already know how to read input into a handy tree data structure and walk it. You can skip ahead to Chapter 6, *Tracking and Identifying Program Symbols*,on page 129 and Chapter 7, *Managing Symbol Tables for Data Aggregates*,on page 153, which describe how to build symbol tables. Symbol tables answer the question "What is *x*?" for some input symbol *x*. They are necessary data structures for the patterns in Chapter 8, *Enforcing Static Typing Rules*,on page 179, for example.

More advanced readers might want to jump directly to Chapter 9, *Building High-Level Interpreters*,on page 217 and Chapter 12, *Generating DSLs with Templates*,on page 307. If you really know what you're doing, you can skip around the book looking for patterns of interest. The truly impatient can grab a sample implementation from a pattern and use it as a kernel for a new language (relying on the book for explanations).

If you bought the e-book version of this book, you can click the gray boxes above the code samples to download code snippets directly. If you'd like to participate in conversations with me and other readers, you can do so at the web page for this book (http://www.pragprog.com/titles/tpdsl) or on the ANTLR user's list (http://www.antlr.org/support.html). You can also post book errata and download all the source code on the book's web page.

Languages and Tools Used in This Book

The code snippets and implementations in this book are written in Java, but their substance applies equally well to any other general programming language. I had to pick a single programming language for consistency. Java is a good choice because it's widely used in industry (http://langpop.com and http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html). Remember, this book is about design patterns, not "language recipes." You can't just download a pattern's sample implementation and apply it to your problem without modification.

We'll use state-of-the-art language tools wherever possible in this book. For example, to recognize (parse) input phrases, we'll use a parser generator (well, that is, after we learn how to build parsers manually in Chapter 2, *Basic Parsing Patterns*,on page 19). It's no fair using a parser generator until you know how parsers work. That'd be like using a calculator before learning to

do arithmetic. Similarly, once we know how to build tree walkers by hand, we can let a tool build them for us.

In this book, we'll use ANTLR extensively. ANTLR is a parser generator and tree walker generator that I've honed over the past two decades while building language applications. I could have used any similar language tool, but I might as well use my own. My point is that this book is not about ANTLR itself—it's about the design patterns common to most language applications. The code samples merely help you to understand the patterns.

We'll also use a template engine called StringTemplate a lot in Chapter 12, *Generating DSLs with Templates*,on page 307 to generate output. StringTemplate is like an "unparser generator," and templates are like output grammar rules. The alternative to a template engine would be to use an unstructured blob of generation logic interspersed with print statements.

You'll be able to follow the patterns in this book even if you're not familiar with ANTLR and StringTemplate. Only the sample implementations use them. To get the most out of the patterns, though, you should walk through the sample implementations. To really understand them, it's a good idea to learn more about the ANTLR project tools. You'll get a taste in Section 4.3, *Quick Introduction to ANTLR*,on page 81. You can also visit the website to get documentation and examples or purchase *The Definitive ANTLR Reference [Par07]* (shameless plug).

One way or another, you're going to need language tools to implement languages. You'll have no problem transferring your knowledge to other tools after you finish this book. It's like learning to fly—you have no choice but to pick a first airplane. Later, you can move easily to another airplane. Gaining piloting skills is the key, not learning the details of a particular aircraft cockpit.

I hope this book inspires you to learn about languages and motivates you to build domain-specific languages (DSLs) and other language tools to help fellow programmers.

Terence Parr
parrt@cs.usfca.edu
December 2009

Part I

Getting Started with Parsing

CHAPTER 1

Language Applications Cracked Open

In this first part of the book, we're going to learn how to recognize computer languages. (A *language* is just a set of valid sentences.) Every language application we look at will have a parser (recognizer) component, unless it's a pure code generator.

We can't just jump straight into the patterns, though. We need to see how everything fits together first. In this chapter, we'll get an architectural overview and then tour the patterns at our disposal. Finally, we'll look at the guts of some sample language applications to see how they work and how they use patterns.

1.1 The Big Picture

Language applications can be very complicated beasts, so we need to break them down into bite-sized components. The components fit together into a multistage pipeline that analyzes or manipulates an input stream. The pipeline gradually converts an input sentence (valid input sequence) to a handy internal data structure or translates it to a sentence in another language.

We can see the overall data flow within the pipeline in Figure 1, *The multistage pipeline of a language application*,on page 5. The basic idea is that a reader recognizes input and builds an *intermediate representation* (IR) that feeds the rest of the application. At the opposite end, a generator emits output based upon the IR and what the application learned in the intermediate stages. The intermediate stages form the *semantic analyzer* component. Loosely speaking, semantic analysis figures out what the input means (anything beyond syntax is called the *semantics*).

The kind of application we're building dictates the stages of the pipeline and how we hook them together. There are four broad application categories:

- *Reader*: A reader builds a data structure from one or more input streams. The input streams are usually text but can be binary data as well. Examples include configuration file readers, program analysis tools such as a method cross-reference tool, and class file loaders.
- *Generator*: A generator walks an internal data structure and emits output. Examples include object-to-relational database mapping tools, object serializers, source code generators, and web page generators.
- *Translator* or *Rewriter*: A translator reads text or binary input and emits output conforming to the same or a different language. It is essentially a combined reader and generator. Examples include translators from extinct programming languages to modern languages, wiki to HTML translators, refactorers, profilers that instrument code, log file report generators, pretty printers, and macro preprocessors. Some translators, such as assemblers and compilers, are so common that they warrant their own subcategories.
- *Interpreter*: An interpreter reads, decodes, and executes instructions. Interpreters range from simple calculators and POP protocol servers all the way up to programming language implementations such as those for Java, Ruby, and Python.

1.2 A Tour of the Patterns

This section is a road map of this book's 31 language implementation patterns. Don't worry if this quick tour is hard to digest at first. The fog will clear as we go through the book and get acquainted with the patterns.

Parsing Input Sentences

Reader components use the patterns discussed in Chapter 2, *Basic Parsing Patterns*,on page 19 and Chapter 3, *Enhanced Parsing Patterns*,on page 47 to *parse* (recognize) input structures. There are five alternative parsing patterns between the two chapters. Some languages are tougher to parse than others, and so we need parsers of varying strength. The trade-off is that the stronger parsing patterns are more complicated and sometimes a bit slower.

We'll also explore a little about grammars (formal language specifications) and figure out exactly how parsers recognize languages. Pattern 1, *Mapping Grammars to Recursive-Descent Recognizers*,on page 27 shows us how to

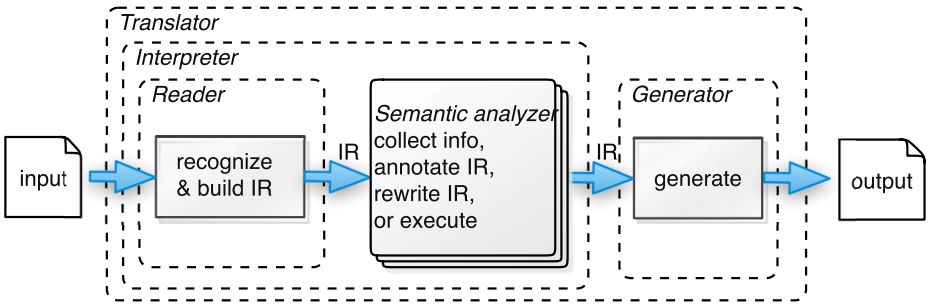


Figure 1—The multistage pipeline of a language application

convert grammars to hand-built parsers. ANTLR[1] (or any similar parser generator) can do this conversion automatically for us, but it's a good idea to familiarize ourselves with the underlying patterns.

The most basic reader component combines Pattern 2, *LL(1) Recursive-Descent Lexer*,on page 31 together with Pattern 3, *LL(1) Recursive-Descent Parser*,on page 36 to recognize sentences. More complicated languages will need a stronger parser, though. We can increase the recognition strength of a parser by allowing it to look at more of the input at once (Pattern 4, *LL(k) Recursive-Descent Parser*,on page 41).

When things get really hairy, we can only distinguish sentences by looking at an entire sentence or phrase (subsentence) using Pattern 5, *Backtracking Parser*,on page 53. Backtracking's strength comes at the cost of slow execution speed. With some tinkering, however, we can dramatically improve its efficiency. We just need to save and reuse some partial parsing results with Pattern 6, *Memoizing Parser*,on page 60.

For the ultimate parsing power, we can resort to Pattern 7, *Predicated Parser*,on page 65. A predicated parser can alter the normal parsing flow based upon run-time information. For example, input `T(i)` can mean different things depending on how we defined `T` previously. A predicate parser can look up `T` in a dictionary to see what it is.

Besides tracking input symbols like `T`, a parser can execute actions to perform a transformation or do some analysis. This approach is usually too simplistic for most applications, though. We'll need to make multiple passes over the input. These passes are the stages of the pipeline that are beyond the reader component.

1. http://www.antlr.org

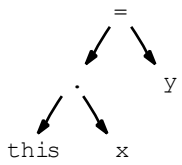
Constructing Trees

Rather than repeatedly parsing the input text in every stage, we'll construct an IR. The IR is a highly processed version of the input text that's easy to traverse. The nodes or elements of the IR are also ideal places to squirrel away information for use by later stages. In Chapter 4, *Building Intermediate Form Trees*,on page 71, we'll discuss why we build trees and how they encode essential information from the input.

The nature of an application dictates what kind of data structure we use for the IR. Compilers require a highly specialized IR that is very low level (elements of the IR correspond very closely with machine instructions). Because we're not focusing on compilers in this book, though, we'll generally use a higher-level tree structure.

The first tree pattern we'll look at is Pattern 8, *Parse Tree*,on page 87. Parse trees are pretty "noisy," though. They include a record of the rules used to recognize the input, not just the input itself. Parse trees are useful primarily for building syntax-highlighting editors. For implementing source code analyzers, translators, and the like, we'll build *abstract syntax trees* (ASTs) because they are easier to work with.

An AST has a node for every important token and uses operators as subtree roots. For example, the AST for assignment statement `this.x=y;` is as follows:



The AST implementation pattern you pick depends on how you plan on traversing the AST (Chapter 4, *Building Intermediate Form Trees*,on page 71 discusses AST construction in detail).

Pattern 9, *Homogeneous AST*,on page 91 is as simple as you can get. It uses a single object type to represent every node in the tree. Homogeneous nodes also have to represent specific children by position within a list rather than with named node fields. We call that a *normalized child list*.

If we need to store different data depending on the kind of tree node, we need to introduce multiple node types with Pattern 10, *Normalized Heterogeneous AST*,on page 93. For example, we might want different node types for addition operator nodes and variable reference nodes. When building heterogeneous

node types, it's common practice to track children with fields rather than lists (Pattern 11, *Irregular Heterogeneous AST*,on page 96).

Walking Trees

Once we've got an appropriate representation of our input in memory, we can start extracting information or performing transformations. To do that, we need to traverse the IR (AST, in our case). There are two basic approaches to tree walking. Either we embed methods within each node class (Pattern 12, *Embedded Heterogeneous Tree Walker*,on page 111) or we encapsulate those methods in an external visitor (Pattern 13, *External Tree Visitor*,on page 113). The external visitor is nice because it allows us to alter tree-walking behavior without modifying node classes.

Rather than build external visitors manually, though, we can automate visitor construction just like we can automate parser construction. To recognize tree structures, we'll use Pattern 14, *Tree Grammar*,on page 117 or Pattern 15, *Tree Pattern Matcher*,on page 120. A tree grammar describes the entire structure of all valid trees, whereas a tree pattern matcher lets us focus on just those subtrees we care about. You'll use one or more of these tree walkers to implement the next stages in the pipeline.

Figuring Out What the Input Means

Before we can generate output, we need to analyze the input to extract bits of information relevant to generation (semantic analysis). Language analysis is rooted in a fundamental question: for a given symbol reference `x`, what is it? Depending on the application, we might need to know whether it's a variable or method, what type it is, or where it's defined. To answer these questions, we need to track all input symbols using one of the *symbol tables* in Chapter 6, *Tracking and Identifying Program Symbols*,on page 129 or Chapter 7, *Managing Symbol Tables for Data Aggregates*,on page 153. A symbol table is just a dictionary that maps symbols to their definitions.

The semantic rules of your language dictate which symbol table pattern to use. There are four common kinds of scoping rules: languages with a single scope, nested scopes, C-style `struct` scopes, and class scopes. You'll find the associated implementations in Pattern 16, *Symbol Table for Monolithic Scope*,on page 139, Pattern 17, *Symbol Table for Nested Scopes*,on page 144, Pattern 18, *Symbol Table for Data Aggregates*,on page 159, and Pattern 19, *Symbol Table for Classes*,on page 164.

Languages such as Java, C#, and C++ have a ton of semantic compile-time rules. Most of these rules deal with type compatibility between operators or

assignment statements. For example, we can't multiply a string by a class name. Chapter 8, *Enforcing Static Typing Rules*,on page 179 describes how to compute the types of all expressions and then check operations and assignments for type compatibility. For non-object-oriented languages like C, we'd apply Pattern 22, *Enforcing Static Type Safety*,on page 198. For object-oriented languages like C++ or Java, we'd apply Pattern 23, *Enforcing Polymorphic Type Safety*,on page 206. To make these patterns easier to absorb, we'll break out some of the necessary infrastructure in Pattern 20, *Computing Static Expression Types*,on page 182 and Pattern 21, *Automatic Type Promotion*,on page 190.

If you're building a reader like a configuration file reader or Java .class file reader, your application pipeline would be complete at this point. To build an interpreter or translator, though, we have to add more stages.

Interpreting Input Sentences

Interpreters execute instructions stored in the IR but usually need other data structures too, like a symbol table. Chapter 9, *Building High-Level Interpreters*,on page 217 describes the most common interpreter implementation patterns, including Pattern 24, *Syntax-Directed Interpreter*,on page 222, Pattern 25, *Tree-Based Interpreter*,on page 227, Pattern 27, *Stack-Based Bytecode Interpreter*,on page 256, and Pattern 28, *Register-Based Bytecode Interpreter*,on page 264. From a capability standpoint, the interpreter patterns are equivalent (or could be made equally powerful). The differences between them lie in the instruction set, execution efficiency, interactivity, ease-of-use, and ease of implementation.

Translating One Language to Another

Rather than interpreting a computer language, we can translate programs to another language (at the extreme, compilers translate high-level programs down to machine code). The final component of any translator is a generator that emits structured text or binary. The output is a function of the input and the results of semantic analysis. For simple translations, we can combine the reader and generator into a single pass using Pattern 29, *Syntax-Directed Translator*,on page 291. Generally, though, we need to decouple the order in which we compute output phrases from the order in which we emit output phrases. For example, imagine reversing the statements of a program. We can't generate the first output statement until we've read the final input statement. To decouple input and output order, we'll use a model-driven approach. (See Chapter 11, *Translating Computer Languages*,on page 275.)

Because generator output always conforms to a language, it makes sense to use a formal language tool to emit structured text. What we need is an "unparser" called a *template engine*. There are many excellent template engines out there but, for our sample implementations, we'll use StringTemplate.[2] (See Chapter 12, *Generating DSLs with Templates*,on page 307.)

So, that's how patterns fit into the overall language implementation pipeline. Before getting into them, though, it's worth investigating the architecture of some common language applications. It'll help keep everything in perspective as you read the patterns chapters.

1.3 Dissecting a Few Applications

Language applications are a bit like fractals. As you zoom in on their architecture diagrams, you see that their pipeline stages are themselves multistage pipelines. For example, though we see compilers as black boxes, they are actually deeply nested pipelines. They are so complicated that we have to break them down into lots of simpler components. Even the individual top-level components are pipelines. Digging deeper, the same data structures and algorithms pop up across applications and stages.

This section dissects a few language applications to expose their architectures. We'll look at a bytecode interpreter, a bug finder (source code analyzer), and a C/C++ compiler. The goal is to emphasize the architectural similarity between applications and even between the stages in a single application. The more you know about existing language applications, the easier it'll be to design your own. Let's start with the simplest architecture.

Bytecode Interpreter

An *interpreter* is a program that executes other programs. In effect, an interpreter simulates a hardware processor in software, which is why we call them *virtual machines*. An interpreter's instruction set is typically pretty low level but higher level than raw machine code. We call the instructions *bytecodes* because we can represent each instruction with a unique integer code from 0..255 (a byte's range).

We can see the basic architecture of a bytecode interpreter in Figure 2, *Bytecode interpreter pipeline*,on page 10. A reader loads the bytecodes from a file before the interpreter can start execution. To execute a program, the interpreter uses a *fetch-decode-execute cycle*. Like a real processor, the

2. http://www.stringtemplate.org

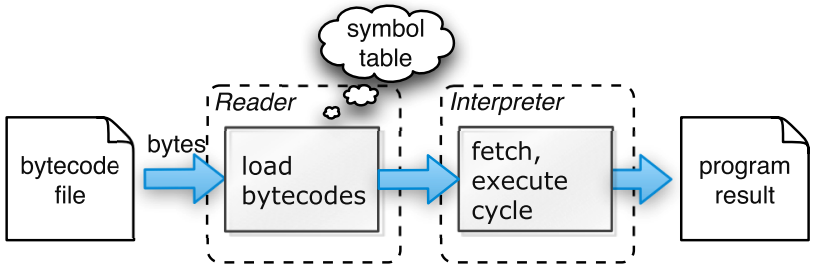


Figure 2—Bytecode interpreter pipeline

interpreter has an instruction pointer that tracks which instruction to execute next. Some instructions move data around, some move the instruction pointer (branches and calls), and some emit output (which is how we get the program result). There are a lot of implementation details, but this gives you the basic idea.

Languages with bytecode interpreter implementations include Java, Lua,[3] Python, Ruby, C#, and Smalltalk.[4] Lua uses Pattern 28, *Register-Based Bytecode Interpreter*,on page 264, but the others use Pattern 27, *Stack-Based Bytecode Interpreter*,on page 256. Prior to version 1.9, Ruby used something akin to Pattern 25, *Tree-Based Interpreter*,on page 227.

Java Bug Finder

Let's move all the way up to the source code level now and crack open a Java bug finder application. To keep things simple, we'll look for just one kind of bug called *self-assignment*. Self-assignment is when we assign a variable to itself. For example, the `setX()` method in the following `Point` class has a useless self-assignment because `this.x` and `x` refer to the same field `x`:

```
class Point {
    int x,y;
    void setX(int y) { this.x = x; } // oops! Meant setX(int x)
    void setY(int y) { this.y = y; }
}
```

The best way to design a language application is to start with the end in mind. First, figure out what information you need in order to generate the output. That tells you what the final stage before the generator computes. Then figure out what that stage needs and so on all the way back to the reader.

3. http://www.lua.org
4. http://en.wikipedia.org/wiki/Smalltalk_programming_language

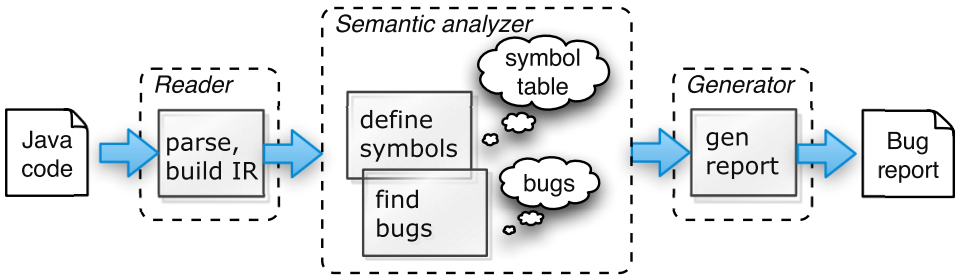


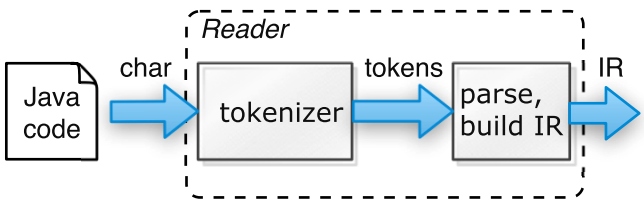
Figure 3—Source-level bug finder pipeline

For our bug finder, we need to generate a report showing all self-assignments. To do that, we need to find all assignments of the form this.x = x and flag those that assign to themselves. To do that, we need to figure out (*resolve*) to which entity this.x and x refer. That means we need to track all symbol definitions using a symbol table like Pattern 19, *Symbol Table for Classes*,on page 164. We can see the pipeline for our bug finder in Figure 3, *Source-level bug finder pipeline*,on page 11.

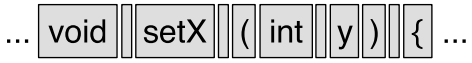
Now that we've identified the stages, let's walk the information flow forward. The parser reads the Java code and builds an intermediate representation that feeds the semantic analysis phases. To parse Java, we can use Pattern 2, *LL(1) Recursive-Descent Lexer*,on page 31, Pattern 4, *LL(k) Recursive-Descent Parser*,on page 41, Pattern 5, *Backtracking Parser*,on page 53, and Pattern 6, *Memoizing Parser*,on page 60. We can get away with building a simple IR: Pattern 9, *Homogeneous AST*,on page 91.

The semantic analyzer in our case needs to make two passes over the IR. The first pass defines all the symbols encountered during the walk. The second pass looks for assignment patterns whose left-side and right-side resolve to the same field. To find symbol definitions and assignment tree patterns, we can use Pattern 15, *Tree Pattern Matcher*,on page 120. Once we have a list of self-assignments, we can generate a report.

Let's zoom in a little on the reader:



Most text readers use a two-stage process. The first stage breaks up the character stream into vocabulary symbols called *tokens*. The parser feeds off these tokens to check syntax. In our case, the tokenizer (or *lexer*) yields a stream of vocabulary symbols like this:



As the parser checks the syntax, it builds the IR. We have to build an IR in this case because we make multiple passes over the input. Retokenizing and reparsing the text input for every pass is inefficient and makes it harder to pass information between stages. Multiple passes also support forward references. For example, we want to be able to see field `x` even if it's defined after method `setX()`. By defining all symbols first, before trying to resolve them, our bug-finding stage sees `x` easily.

Now let's jump to the final stage and zoom in on the generator. Since we have a list of bugs (presumably a list of `Bug` objects), our generator can use a simple `for` loop to print out the bugs. For more complicated reports, though, we'll want to use a template. For example, if we assume that `Bug` has fields `file`, `line`, and `fieldname`, then we can use the following two StringTemplate template definitions to generate a report (we'll explore template syntax in Chapter 12, *Generating DSLs with Templates*,on page 307).

```
report(bugs) ::= "<bugs:bug()>" // apply template bug to each bug object
bug(b) ::= "bug: <b.file>:<b.line> self assignment to <b.fieldname>"
```

All we have to do is pass the list of `Bug` objects to the `report` template as attribute `bugs`, and StringTemplate does the rest.

There's another way to implement this bug finder. Instead of doing all the work to read Java source code and populate a symbol table, we can leverage the functionality of the `javac` Java compiler, as we'll see next.

Java Bug Finder Part Deux

The Java compiler generates `.class` files that contain serialized versions of a symbol table and AST. We can use Byte Code Engineering Library (BCEL)[5] or another class file reader to load `.class` files instead of building a source code reader (the fine tool FindBugs[6] uses this approach). We can see the pipeline for this approach in Figure 4, *Java bug finder pipeline feeding off class files*,on page 13.

5. http://jakarta.apache.org/bcel/
6. http://findbugs.sourceforge.net/

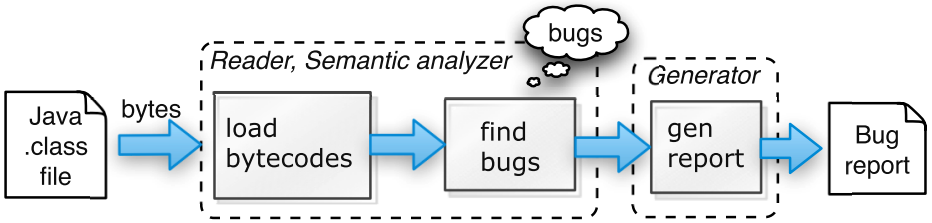


Figure 4—Java bug finder pipeline feeding off .class files

The overall architecture is roughly the same as before. We have just short-circuited the pipeline a little bit. We don't need a source code parser, and we don't need to build a symbol table. The Java compiler has already resolved all symbols and generated bytecode that refers to unique program entities. To find self-assignment bugs, all we have to do is look for a particular bytecode sequence. Here is the bytecode for method setX():

```
0:   aload_0      // push 'this' onto the stack
1:   aload_0      // push 'this' onto the stack
2:   getfield #2; // push field this.x onto the stack
5:   putfield #2; // store top of stack (this.x) into field this.x
8:   return
```

The #2 operand is an offset into a symbol table and uniquely identifies the x (field) symbol. In this case, the bytecode clearly gets and puts the same field. If this.x referred to a different field than x, we'd see different symbol numbers as operands of getfield and putfield.

Now, let's look at the compilation process that feeds this bug finder. javac is a compiler just like a traditional C compiler. The only difference is that a C compiler translates programs down to instructions that run natively on a particular CPU.

C Compiler

A C compiler looks like one big program because we use a single command to launch it (via cc or gcc on UNIX machines). Although the actual C compiler is the most complicated component, the C compilation process has lots of players.

Before we can get to actual compilation, we have to preprocess C files to handle includes and macros. The preprocessor spits out pure C code with some line number directives understood by the compiler. The compiler munches on that for a while and then spits out assembly code (text-based

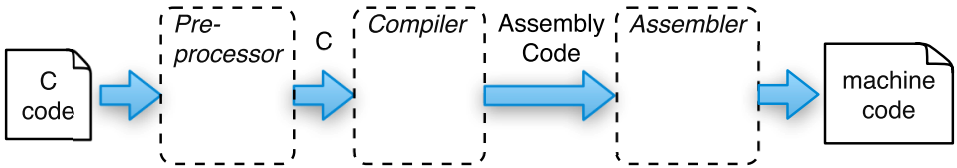


Figure 5—C compilation process pipeline

human-readable machine code). A separate assembler translates the assembly code to binary machine code. With a few command-line options, we can expose this pipeline.

Let's follow the pipeline (shown in Figure 5, *C compilation process pipeline*,on page 14) for the C function in file t.c:

```
void f() { ; }
```

First we preprocess t.c:

```
$ cpp t.c tmp.c          # preprocess t.c, put output into tmp.c
$
```

That gives us the following C code:

```
# 1 "t.c"                // line information generated by preprocessor
# 1 "<built-in>"         // it's not C code per se
# 1 "<command line>"
# 1 "t.c"
void f() { ; }
```

If we had included stdio.h, we'd see a huge pile of stuff in front of f(). To compile tmp.c down to assembly code instead of all the way to machine code, we use option -S. The following session compiles and prints out the generated assembly code:

```
$ gcc -S tmp.c              # compile tmp.c to tmp.s
$ cat tmp.s                 # print assembly code to standard output
        .text
.globl _f
_f:                         ; define function f
        pushl   %ebp        ; do method bookkeeping
        movl    %esp, %ebp  ; you can ignore this stuff
        subl    $8, %esp
        leave               ; clean up stack
        ret                 ; return to invoking function
        .subsections_via_symbols
$
```

To assemble tmp.s, we run as to get the object file tmp.o:

```
$ as -o tmp.o tmp.s          # assemble tmp.s to tmp.o
$ ls tmp.*
tmp.c   tmp.o   tmp.s
$
```

Now that we know about the overall compilation process, let's zoom in on the pipeline inside the C compiler itself.

The main components are highlighted in Figure 6, *Isolated C compiler application pipeline*,on page 16. Like other language applications, the C compiler has a reader that parses the input and builds an IR. On the other end, the generator traverses the IR, emitting assembly instructions for each subtree. These components (the *front end* and *back end*) are not the hard part of a compiler.

All the scary voodoo within a compiler happens inside the semantic analyzer and optimizer. From the IR, it has to build all sorts of extra data structures in order to produce an efficient version of the input C program in assembly code. Lots of set and graph theory algorithms are at work. Implementing these complicated algorithms is challenging. If you'd like to dig into compilers, I recommend the famous "Dragon" book: *Compilers: Principles, Techniques, and Tools [ALSU06]* (Second Edition).

Rather than build a complete compiler, we can also leverage an existing compiler. In the next section, we'll see how to implement a language by translating it to an existing language.

Leveraging a C Compiler to Implement C++

Imagine you are Bjarne Stroustrup, the designer and original implementer of C++. You have a cool idea for extending C to have classes, but you're faced with a mammoth programming project to implement it from scratch.

To get C++ up and running in fairly short order, Stroustrup simply reduced C++ compilation to a known problem: C compilation. In other words, he built a C++ to C translator called cfront. He didn't have to build a compiler at all. By generating C, his nascent language was instantly available on any machine with a C compiler. We can see the overall C++ application pipeline in Figure 7, *C++ (cfront) compilation process pipeline*,on page 16. If we zoomed in on cfront, we'd see yet another reader, semantic analyzer, and generator pipeline.

As you can see, language applications are all pretty similar. Well, at least they all use the same basic architecture and share many of the same components. To implement the components, they use a lot of the same patterns.

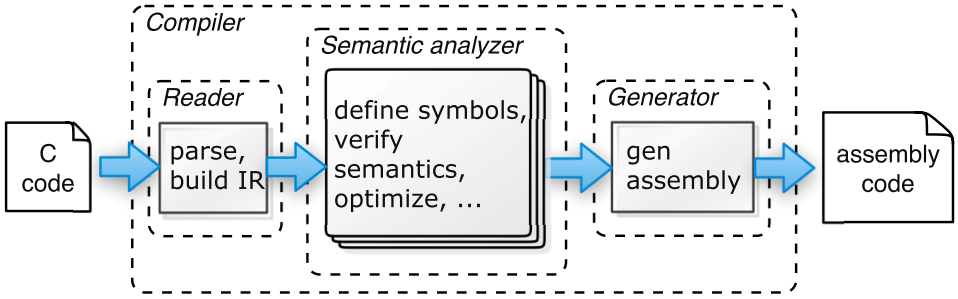


Figure 6—Isolated C compiler application pipeline

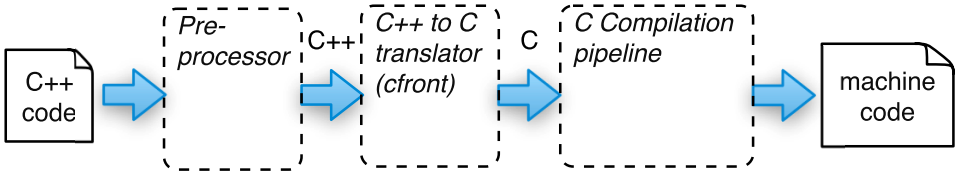


Figure 7—C++ (cfront) compilation process pipeline

Before moving on to the patterns in the subsequent chapters, let's get a general sense of how to hook them together into our own applications.

1.4 Choosing Patterns and Assembling Applications

I chose the patterns in this book because of their importance and how often you'll find yourself using them. From my own experience and from listening to the chatter on the ANTLR interest list, we programmers typically do one of two things. Either we implement DSLs or we process and translate general-purpose programming languages. In other words, we tend to implement graphics and mathematics languages, but very few of us build compilers and interpreters for full programming languages. Most of the time, we're building tools to refactor, format, compute software metrics, find bugs, instrument, or translate them to another high-level language.

If we're not building implementations for general-purpose programming languages, you might wonder why I've included some of the patterns I have. For example, all compiler textbooks talk about symbol table management and computing the types of expressions. This book also spends roughly 20 percent of the page count on those subjects. The reason is that some of the patterns

we'd need to build a compiler are also critical to implementing DSLs and even just processing general-purpose languages. Symbol table management, for example, is the bedrock of most language applications you'll build. Just as a parser is the key to analyzing the syntax, a symbol table is the key to understanding the semantics (meaning) of the input. In a nutshell, syntax tells us what to do, and semantics tells us what to do it to.

As a language application developer, you'll be faced with a number of important decisions. You'll need to decide which patterns to use and how to assemble them to build an application. Fortunately, it's not as hard as it seems at first glance. The nature of an application tells us a lot about which patterns to use, and, amazingly, only two basic architectures cover the majority of language applications.

Organizing the patterns into groups helps us pick the ones we need. This book organizes them more or less according to Figure 1, *The multistage pipeline of a language application*,on page 5. We have patterns for reading input (part I), analyzing input (part II), interpreting input (part III), and generating output (part IV). The simplest applications use patterns from part I, and the most complicated applications need patterns from I, II, and III or from I, II, and IV. So, if all we need to do is load some data into memory, we pick patterns from part I. To build an interpreter, we need patterns to read the input and at least a pattern from part III to execute commands. To build a translator, we again need patterns to parse the input, and then we need patterns from part IV to generate output. For all but the simplest languages, we'll also need patterns from part II to build internal data structures and analyze the input.

The most basic architecture combines lexer and parser patterns. It's the heart of Pattern 24, *Syntax-Directed Interpreter*,on page 222 and Pattern 29, *Syntax-Directed Translator*,on page 291. Once we recognize input sentences, all we have to do is call a method that executes or translates them. For an interpreter, this usually means calling some implementation function like `assign()` or `drawLine()`. For a translator, it means printing an output statement based upon symbols from the input sentence.

The other common architecture creates an AST from the input (via tree construction actions in the parser) instead of trying to process the input on the fly. Having an AST lets us sniff the input multiple times without having to reparse it, which would be pretty inefficient. For example, Pattern 25, *Tree-Based Interpreter*,on page 227 revisits AST nodes all the time as it executes `while` loops, and so on.

The AST also gives us a convenient place to store information that we compute in the various stages of the application pipeline. For example, it's a good idea to annotate the AST with pointers into the symbol table. The pointers tell us what kind of symbol the AST node represents and, if it's an expression, what its result type is. We'll explore such annotations in Chapter 6, *Tracking and Identifying Program Symbols*,on page 129 and Chapter 8, *Enforcing Static Typing Rules*,on page 179.

Once we've got a suitable AST with all the necessary information in it, we can tack on a final stage to get the output we want. If we're generating a report, for example, we'd do a final pass over the AST to collect and print whatever information we need. If we're building a translator, we'd tack on a generator from Chapter 11, *Translating Computer Languages*,on page 275 or Chapter 12, *Generating DSLs with Templates*,on page 307. The simplest generator walks the AST and directly prints output statements, but it works only when the input and output statement orders are the same. A more flexible strategy is to construct an output model composed of strings, templates, or specialized output objects.

Once you have built a few language applications, you will get a feel for whether you need an AST. If I'm positive I can just bang out an application with a parser and a few actions, I'll do so for simplicity reasons. When in doubt, though, I build an AST so I don't code myself into a corner.

Now that we've gotten some perspective, we can begin our adventure into language implementation.

CHAPTER 2

Basic Parsing Patterns

Language recognition is a critical step in just about any language application. To interpret or translate a phrase, we first have to recognize what kind of phrase it is (sentences are made up of phrases). Once we know that a phrase is an assignment or function call, for example, we can act on it. To recognize a phrase means two things. First, it means we can distinguish it from the other constructs in that language. And, second, it means we can identify the elements and any substructures of the phrase. For example, if we recognize a phrase as an assignment, we can identify the variable on the left of the = and the expression substructure on the right. The act of recognizing a phrase by computer is called *parsing*.

This chapter introduces the most common parser design patterns that you will need to build recognizers by hand. There are multiple parser design patterns because certain languages are harder to parse than others. As usual, there is a trade-off between parser simplicity and parser strength. Extremely complex languages like C++ typically require less efficient but more powerful parsing strategies. We'll talk about the more powerful parsing patterns in the next chapter. For now, we'll focus on the following basic patterns to get up to speed:

- Pattern 1, *Mapping Grammars to Recursive-Descent Recognizers*,on page 27. This pattern tells us how to convert a grammar (formal language specification) to a hand-built parser. It's used by the next three patterns.
- Pattern 2, *LL(1) Recursive-Descent Lexer*,on page 31. This pattern breaks up character streams into tokens for use by the parsers defined in the subsequent patterns.
- Pattern 3, *LL(1) Recursive-Descent Parser*,on page 36. This is the most well-known recursive-descent parsing pattern. It only needs to look at

the current input symbol to make parsing decisions. For each rule in a grammar, there is a parsing method in the parser.

- Pattern 4, *LL(k) Recursive-Descent Parser*,on page 41. This pattern augments an *LL(1)* recursive-descent parser so that it can look multiple symbols ahead (up to some fixed number *k*) in order to make decisions.

Before jumping into the parsing patterns, this chapter provides some background material on language recognition. Along the way, we will define some important terms and learn about grammars. You can think of grammars as functional specifications or design documents for parsers. To build a parser, we need a guiding specification that precisely defines the language we want to parse.

Grammars are more than designs, though. They are actually executable "programs" written in a domain-specific language (DSL) specifically designed for expressing language structures. Parser generators such as ANTLR can automatically convert grammars to parsers for us. In fact, ANTLR mimics what we'd build by hand using the design patterns in this chapter and the next.

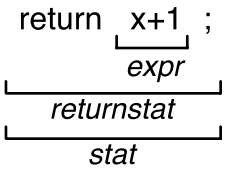
After we get a good handle on building parsers by hand, we'll rely on grammars throughout the examples in the rest of the book. Grammars are often 10 percent the size of hand-built recognizers and provide more robust solutions. The key to understanding ANTLR's behavior, though, lies in these parser design patterns. If you have a solid background in computer science or already have a good handle on parsing, you can probably skip this chapter and the next.

Let's get started by figuring out how to identify the various substructures in a phrase.

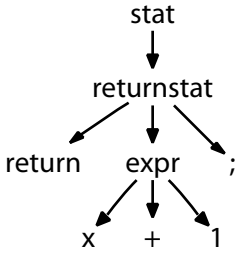
2.1 Identifying Phrase Structure

In elementary school, we all learned (and probably forgot) how to identify the parts of speech in a sentence like *verb* and *noun*. We can do the same thing with computer languages (we call it *syntax analysis*). Vocabulary symbols (*tokens*) play different roles like *variable* and *operator*. We can even identify the role of token subsequences like *expression*.

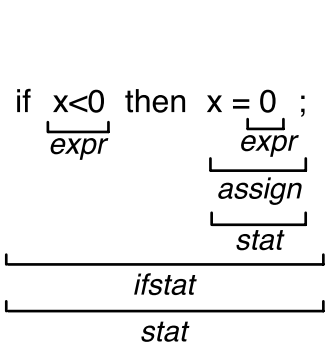
Take `return x+1;`, for example. Sequence `x+1` plays the role of an expression and the entire phrase is a return statement, which is also a kind of statement. If we represent that visually, we get a sentence diagram of sorts:

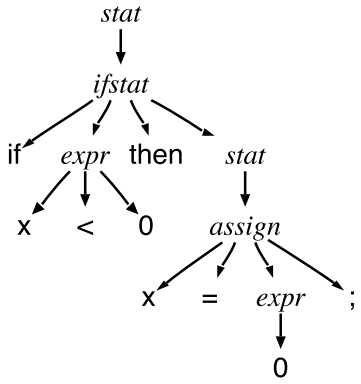


Flip that over, and you get what we call a *parse tree*:



Tokens hang from the parse tree as leaf nodes, while the interior nodes identify the phrase substructures. The actual names of the substructures aren't important as long as we know what they mean. For a more complicated example, take a look at the substructures and parse tree for an `if` statement:





Parse trees are important because they tell us everything we need to know about the syntax (*structure*) of a phrase. To parse, then, is to conjure up a two-dimensional parse tree from a flat token sequence.

2.2 Building Recursive-Descent Parsers

A parser checks whether a sentence conforms to the syntax of a *language*. (A language is just a set of valid sentences.) To verify language membership, a